Accurate and Efficient Reliability Estimation Techniques during ADL-Driven Embedded Processor Design

Zheng Wang†, Kapil Singh‡, Chao Chen†, Anupam Chattopadhyay†
† MPSoC Architectures Research Group, UMIC, RWTH Aachen University, Germany
‡ Department of Electrical Engineering, IIT Kanpur, India
wang@umic.rwth-aachen.de

***Abstract*—The downscaling of technology features has brought the system developers an important design criteria, reliability, into prime consideration. Due to external radiation effects and temperature gradients, the CMOS device is not guaranteed anymore to function flawlessly. On the other hand, admission for errors to occur allows extending the power budget. The power-performance-reliability trade-off compounds the system design challenge, for which efficient design exploration framework is needed. In this work, we present a high-level processor design framework extended with two reliability estimation techniques. First, a simulation-based technique, which allows a generic instruction-set simulator to estimate reliability via high-level fault injection capability. Second, a novel analytical technique, which is based on the reliability model for coarse arithmetic logical operator blocks within a processor instruction. The techniques are tested with a RISC processor and several embedded application kernels. Our results show the efficiency and accuracy of these techniques against a HDL-level reliability estimation framework.**

***Keywords*-Reliability Estimation; High-level Processor Design; Fault Simulation**

I. Introduction

With continuous downscaling of CMOS technology nodes, reliability has become a serious design concern. This is influenced by several trends. First, soft errors caused by external radiation are increasingly reported, even at ground conditions[1]. Second, increasing power dissipation issues reportedly affects design lifetime as well as causing transient errors[2]. Third, by allowing temperature-induced error rates to happen, overcautious guard-banding can be avoided thereby, increasing power budget and performance[3]. Finally, new kinds of innovative fault-based attacks against embedded cryptographic modules[4] make fault-tolerant design important. Fault tolerance is also mandatory for other safety-critical application domains such as biomedical, automotive and infrastructure, where embedded systems are being increasingly used. Naturally, reliability is being considered as a design constraint in late CMOS era and being explored across all the layers of a system by research community[5]. We focus on the reliability estimation of embedded processors, which is increasingly used in modern multiprocessor System-on-Chips (SoCs).

The unreliability of a design can be traced back to two major categories of faults, namely, transient and permanent faults [6]. While *permanent faults* are hard damages caused by extrinsic sources such as manufacturing defects and process variation, *transient faults* temporarily corrupt the data or the output of a combinational circuit. The impact of transient faults increases with the technology and voltage scaling and the increasing complexity of digital systems[7].

The impact of faults can be investigated through simulation. While faults can be simulated accurately only at the circuit level of abstraction, there have been many proposals to inject the fault at high level of abstraction for early performance exploration. Pure software fault injection techniques alter the processor state (memory, register) to simulate a fault. Naturally, it suffers from a restricted view of the microarchitecture. Register-Transfer Level (RTL) fault injection approaches [8] simulate the microarchitecture with more accuracy and therefore, are usually slow and in some cases need repeated compilation. In order to strike a balance, fault injection can be done during instruction-set simulation. Processor instruction set simulators offer different degrees of accuracy and speed trade-off. This is leveraged by creating a fault injection set up based on cycle-accurate instruction-set simulator in [9] and [10]. We propose to extend these by having the reliability estimation flow coupled with an Architecture Description Language(ADL)-based processor design framework. ADLs are dominantly used for modeling, designing and verifying embedded processors. Several ADL-based processor design frameworks are commercially [11], [12] available.

Complementing the simulation methods, analytical methods have also been proposed to investigate behavior of circuits under faults. Mukherjee*et al* [13] introduced the concept of architecturally correct execution (ACE) to compute the vulnerability factors of faulty structures. In [14] the authors performed the ACE analysis to compute architectural vulnerability factors for cache and buffers. Recently, Rehman*et al* [15], [16] extended the ACE concepts to instruction vulnerability analysis and proposed reliability-aware software transformations. We analyse the vulnerability of the instruction by studying the constituent logic blocks and by that possibly connect with the circuit-level reliability analysis [17]. While the instruction vulnerability index model proposed at [15] includes the logical masking effects, the details of derivation of the masking effect are not mentioned. The simulation accuracy is compared with other software-level reliability estimation flows [15].

A. Motivation and Contribution

Processor designers require techniques to accurately and efficiently estimate the reliability so that the software and architecture exploration can be made in early design stage. To increase accuracy, such method has to be assisted by high level fault injection. However, the lack of generic tools for fault

978-3-9815370-0-0/DATE13/ © 2013 EDAA

injection at architectural level enforces the designer to include at least the RTL abstraction in the design space exploration flow or stick to a specific processor design environment, which impedes early design exploration. Furthermore, fault injection and reliability estimation at processor abstraction level focused dominantly at the storage [18], [16].

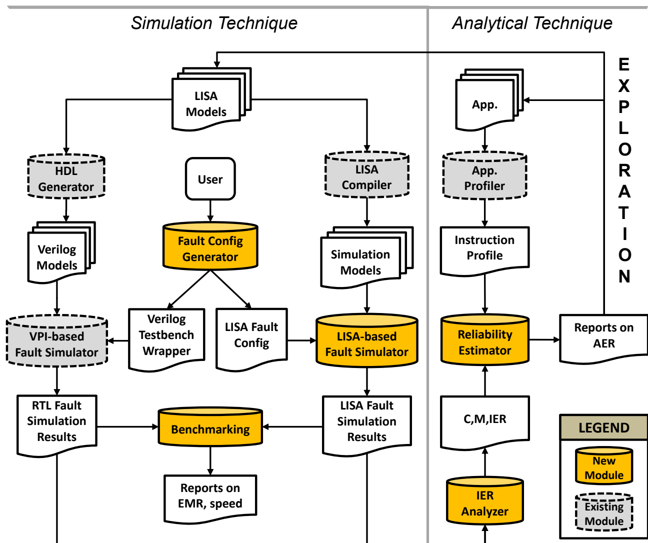

Figure 1. ADL-driven Reliability Estimation Flow

In this paper, we study the option of including reliability as a design constraint early in the design flow based on ADL LISA [11], though our techniques are general enough to be applicable for any high-level processor design environment. In short, our contributions in this paper are first, proposing a generic simulation-based reliability estimation flow, second, an analytical reliability estimation flow and finally to benchmark these against state-of-the-art RTL-based reliability estimation flow for speed and accuracy.

Figure 1 shows our contributions where the novel modules are filled in dark color. The simulation-based reliability estimation technique starts with ADL description of the processor model. After the RTL and LISA simulation models are generated, both models at different abstraction layers are simulated under faults while their corresponding simulation results are benchmarked to show the accuracy and efficiency of the technique. The analytical technique takes the instruction profiling of the target application and fault simulation results at either abstraction layer as inputs. Such results are used to calculate the operation fault properties and Instruction Error Rate (IER) which are then processed by the reliability estimator to predict the Application Error Rate (AER). Users can improve LISA models and target applications to tune the AER, which closes the reliability estimation/exploration loop.

The rest of the paper is organized as following. Section II discusses the simulation-based technique. Section III explains the analytical reliability estimation technique. Section IV presents few case studies with the presented approaches. The paper is concluded and future work is outlined in section V.

II. Simulation Based Technique

The ADL-based fault simulation technique is presented in this section. First, the ADL LISA is briefly introduced. Then the fault injection approach on LISA processor model is discussed. After that the applied method of error evaluation is presented.

A. Brief Overview of LISA

In LISA, the OPERATION is the key element to describe the instruction set, timing and behavior of the processor. One instruction can be hierarchically split into several OPERATIONs, each of which describes one part of the instruction. The OPERATION contains several sections such as CODING, SYNTAX and BEHAVIOR which represent the encoding, syntax and behavior of the instruction respectively. The BEHAVIOR section encloses plain C codes so that arbitrary functionalities can be specified. The C code allows user-defined data types such as bit types.

The declarations for processor resources such as pipeline stages, registers, memories and ports are located in the global RESOURCE section of LISA and they can be accessed from the LISA OPERATION. To describe the microarchitecture, structural information can be added by assigning the operations to different pipeline stages outside the RESOURCE section. This essentially covers the scheduling. The microarchitecture and instruction set architecture (ISA) are completely described by the RESOURCE and OPERATIONs. For further information on LISA language please refer to [19].

B. Fault Injection Method

To represent physical faults which occur in advanced technology nodes, a detailed fault model is prepared in this work including control over fault type (bit-flip, stuck-at), fault injection location and fault injection duration. This framework is designed so as to accommodate new fault models smoothly.

Simulation-based fault injection can be classified as Code Modification (CM) and Simulator Commands (SC) methods. *Saboteurs* and *Mutant*[20][21] represent variants of the CM methods while [8] shows an example of the SC method. CM methods have the advantage in the controllability of fault injection locations but suffers from time-consuming recompilation of the models. In contrast, SC methods realize fault injection without and code modification and recompilation but its effect is limited by the architecturally visible signals during simulation.

In this work, simulated fault injection technique on LISA simulator is developed. To overcome the inefficiencies of the LISA SC method, a hybrid approach which combines both SC and CM methods is adapted. In the instruction set simulator generated by the LISA compiler the reachable signals from the simulator are limited to the processor resources defined in LISA resource sections. However, the local variables which reside in the LISA operations are still inaccessible. To address this problem, additional LISA signals which can be reached through instruction-set simulator through LISA programming interface are declared. These signals help the simulator to change the values of local variables. Such hybrid approach extends the controllability of the LISA SC methods.

C. Error Evaluation Method

The *Error Manifestation Rate* (EMR) is used as the metric for the evaluation of fault simulation. Suppose one fault is injected randomly both in time and location into the device model or a component of such device in each experiment, EMR is defined as the percentage of experiments which detects error on the memory interfaces to which the faults propagate. Mathematically, given the number of total experiments as N_i, $EMR = N_e/N_i$ where N_e is the number of experiments with error detected. Normally, the EMR increases with the duration and number of injected faults. For the same fault duration and number, larger EMR value means less reliability of the faulty component. The interface values are traced and compared with fault-free golden simulation for fault detection.

To facilitate fault injection and evaluation, a graphical user interface is developed to record user defined fault configuration. The configuration file is processed by the fault parser, which generates data structures with fault specific information. Such structures are then scheduled in a fault event queue using LISA application programming interface. When the simulation starts, the simulator detects the occurrence of faults in the event queue based on their injection time and change the corresponding resource states according to fault configuration. The value of traced signals are recorded dynamically for later analysis.

III. Analytical Reliability Estimation Technique

To present the analytical technique we first explain the operation reliability model, which is applied next to calculate instruction error rate. Then by profiling the target application we derive the application error rates. The target processor for this analysis is a pipelined RISC processor model, which is available via [11].

A. Operation Reliability Model

Directed Acyclic Graph (DAG) is used to represent the activation chain of LISA operations. To represent fault injection and error propagation, data flows have to be added in the DAG. Figure 2 shows the data flow graph for the ALU instruction. While the nodes represent LISA operations the edge between them shows the data flow with an individual index and corresponding signal names. When a transient fault is injected into an operation, it needs to first manifest on the operation's output edges and then propagate through following operations until it manifests on the output of the Writeback operation to result in an instruction level error. Notice that not all faults will result in an instruction level error due to logic masking effect. Consequently, the operation error probability and masking probability are proposed to model such process.

Operation error probability C_{op}^{e} is the probability of a detected error on the output edge e of an operation when a fault is injected inside its operation.

Operation masking probability $M_{op}^{e_in,e_out}$ is the probability of a detected error on the output edge e_out of an operation when a fault is injected in its input edge e_in.

Each operation has both C_{op}^{e} and $M_{op}^{e_in,e_out}$ to represent the situation of fault injection on it and error propagation through it respectively. For a particular architecture model, single bit fault

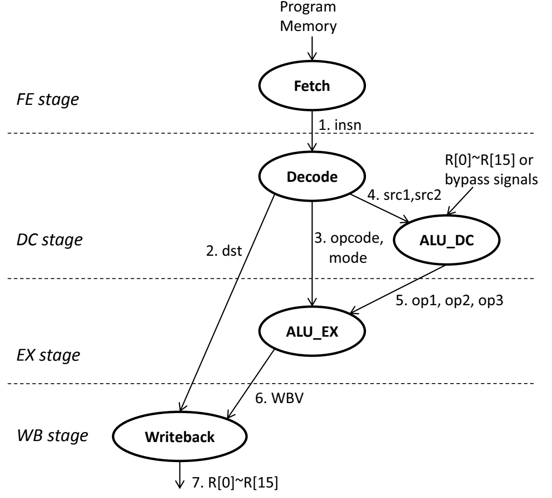

Figure 2. Data flow graph for ALU instruction

is injected through disturbance signals inside of each operation randomly in time and location. By tracing the output edges and comparing the traced value with golden simulation it is easy to get C_{op}^{e} when large number of simulations are performed to counter the randomness. $M_{op}^{e_in,e_out}$ can also be acquired when faults are injected to the input edges while output edges are traced and compared. Theoretical analysis on the data flow graph of combinational logic inside each operation instead of simulation method can also predict its C_{op}^{e} and $M_{op}^{e_in,e_out}$ value similar to [22].

B. Instruction Error Rate

The path error probability is the product of C_{op}^{e} and the $M_{op}^{e_in,e_out}$ of its following operations on the same path from the fault injected operation to the sink operation. The instruction error rate $IER_{insn}^{op_faulty}$ for operation op_faulty and for instruction $insn$ is defined as the summation of all path error probabilities. For example Equation 1 shows the instruction error rate when operation Fetch in Figure 2 is fault injected. The edges in the equation are labeled by their indexes.

$$\begin{aligned} IER_{alu}^{fetch} &= C_{fetch}^{1} M_{decode}^{1,2} M_{writeback}^{2,7} \\ &+ C_{fetch}^{1} M_{decode}^{1,3} M_{alu_ex}^{3,6} M_{writeback}^{6,7} \\ &+ C_{fetch}^{1} M_{decode}^{1,4} M_{alu_dc}^{4,5} M_{alu_ex}^{5,6} M_{writeback}^{6,7} \end{aligned} \quad (1)$$

The method above to calculate the instruction error rate can be applied to all instructions which are defined as a chain of activated operations in LISA. An instruction error can be resulted from a fault injected in each preceding operation in the instruction data flow graph. So that the error rates for a particular instruction constitute a set of $IER_{insn}^{op_faulty}$ where op_faulty is one of the activated operations for $insn$. Besides the operations the edges between them can also be faulty, which resembles the situation when fault is injected on storage resources such as signals and registers. Such resources have essentially both error and masking probabilities equal to one since no masking effect exist for the resources, so that they propagate any encountering fault. In this paper the errors caused by resources are not

considered since we focus on those caused inside combinational logics primarily.

C. Application Error Rate

The application error rate $AER_{app}^{op_faulty}$ represents the error probability when a fault is injected inside operation op_faulty during the execution of a specific application app. When the error rates for all the instructions are known, the application error rate is defined to be the weighted average of all instruction error rates, where the weight of each instruction is its execution counts versus the total instruction counts of the whole application. Figure 3 shows the DAG for all instructions of the RISC processor model. Several instructions which have similar operand behaviors are grouped into the same operation for simplicity. Each instruction corresponds to a path starting from operation Fetch to its sink operations, which interact with resources such as register file or memories. The weights of instructions are labeled pi, which can be acquired from the application profiler. As an example, the application error rate of alu_rrr_ex operation is shown in Equation 2. The summation happens since the operation is on the activation chain of two instructions alu_rrr and alu_rrri.

$$AER_{app}^{alu_rrr_ex} = p_{app}^{1} IER_{alu_rrri}^{alu_rrr_ex} + p_{app}^{2} IER_{alu_rrr}^{alu_rrr_ex} \tag{2}$$

The application error here is detected through the mismatch of instruction results, either committed values to register files or load/store values to memories, with the golden simulation. This provides a conservative estimate of the error rate in program's output, which is normally the value sent by the processor through I/O instructions. The error in the current setup may not lead to an I/O error. This can be caused by several factors. First, the erroneous value committed to architecture registers can be masked by following instructions before I/O access. Second, affected operations which are not activated can be irrelevant to the value finally sent through I/O. Besides, the hardware bypass features in the processor can also silent the interface error since the source of operands can be the bypassed value from pipeline registers instead of architecture registers. In this case an error happened to the writeback value after it is bypassed to later instructions may also not result in an error. However, the proposed analysis offers a fast method to determine to what extent a fault injected operation can potentially influence the program output so that engineers can adopt software or hardware measures to improve system reliability.

IV. Experimental Results

We present two case studies in this section. To demonstrate the effectiveness of our ADL-based simulation technique we first benchmark it with a state-of-the-art HDL-based fault injection. In the second case study we present the results of analytical reliability estimation for the RISC processor.

A. Benchmarking with HDL-based Fault Injection

In [8] a HDL-based fault injection framework is presented, which is based on Verilog programming interface (VPI). The fault injection is realized by scheduling events in the event queue of Verilog simulator. By providing gate-level netlist and

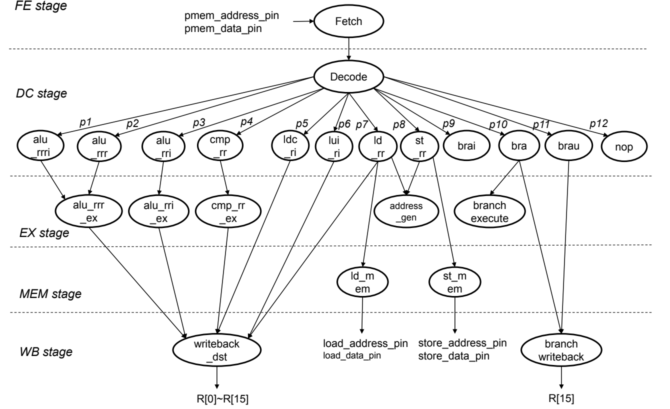

Figure 3. Operation graph for all instructions in RISC processor

standard cell library the framework in [8] can perform gate-level fault simulation. For cycle accurate simulation on ADL level such technology factor is missing. Consequently, we compare our technique with the RTL-level fault simulation provided by their work.

Experiments are carried out on the RISC processor with 5 pipeline stages, which is part of IPs distributed with Processor Designer. The target application is *Sieve of Eratosthenes* which is used to generate prime number within a specified range. After compilation of the LISA descriptions, LISA-based fault injection can be applied directly on the generated processor simulator. To use the HDL-based fault injection framework, the Verilog description is automatically generated from the LISA model. In the following we conduct the comparison with this work with regard to the accuracy and speed.

1) Accuracy: A group of fault configurations are chosen to demonstrate the accuracy in which the duration of single bit-flip fault is changing from 1 to 6 clock cycles. The simulation time is 1200 clock cycles. Each measurement point averages 3000 fault injection experiments for a good approximation.

Figure 4 shows the EMR trends with duration of faults. We see that LISA based fault simulation framework provides similar results as RTL level fault simulation. Differences in absolute values for the same duration are caused by two factors. First, since both injection frameworks generate their faults information independently, two random sets of experiments at LISA and RTL level differ each other with regard to injection time and location. Second, in most cases LISA fault injection gives slightly higher value than RTL injection. This happens because signals which have less effects in fault simulation are missing in LISA description, but are still available for RTL tool to inject fault on.

With regard to individual hardware modules, we find that Fetch unit is most vulnerable among all six modules. This can be easily seen from Figure 3, where Fetch module is on the paths of all operations. So that a fault inside it can potentially influence all instructions. Besides, WRITEBACK module which contains $writeback_dst$ and $branch_writeback$ operations gives lower EMR even though it is on the paths of several instructions. The reason is that during execution, operands of most instructions are bypassed from pipeline registers rather than being obtained

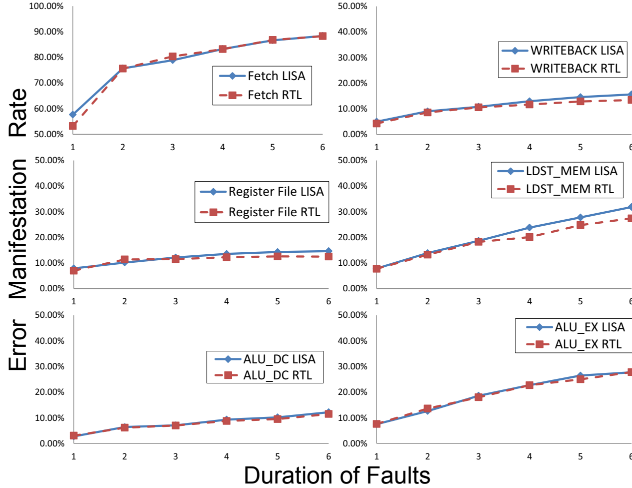

Figure 4. Exemplary EMR with increasing duration of fault

from the register files. Therefore an error in architectural register unnecessarily leads to an error on the memory interface.

2) Speed: Speed is a major metric in the evaluation of any fault injection framework. Table I presents the time required to complete 3000 experiments by both frameworks. Simulations are done on the same host machine and the simulation time for the application is 1200 clock cycles.

Table I
COMPARISON OF FAULT SIMULATION SPEED

Frameworks	**Time duration (3000 exps)**
LISA-based Fault Simulation	935 seconds
HDL-based Fault Simulation	9963 seconds

We can see that LISA-based fault simulation achieves a 10x speed-up compared with the HDL-based one. This is due to the fact that simulations at higher abstraction levels are orders of magnitude faster[23].

B. Analytical Reliability Estimation for RISC Processor

In this section, we present the reliability analysis based on the methodology in Section III. First, the estimation of IER for individual operation is shown. In the next AER is calculated from IER and application dependent weights of instructions. The estimated values are compared with experimental values.

1) IER: A set of testbenches are developed to get individual $IER_{insn}^{op_faulty}$. Each testbench contains the same type of instructions with different modes and random operands. Single bit-flip fault with duration 1 clock cycle targeting a specific operation is then injected during each simulation. Mismatches can be easily detected when both faulty and golden simulations are performed. Each operation specific $IER_{insn}^{op_faulty}$ is obtained from 3000 simulations. The IER can also be derived analytically from Equation 1, where C_{op}^{e} and $M_{op}^{e_in,e_out}$ need to be obtained based on fault simulations. Here we apply the experimental value simply for higher estimation accuracy.

Table II shows $IER_{insn}^{op_faulty}$s of instruction alu_rrr as an example. Table II also shows the application dependent weights of instructions for Sobel. The weights are used to calculate $p \cdot IER$, which constitutes one portion of the AER in Equation 2. Such weights can be obtained directly by the profiling tools of Processor Designer. Note that alu_rrr_dc and alu_rrr_ex operations are subdivided into several modes. This is because different modes of the same instruction type have distinct IERs and weights. The IER among different modes is the weighted average of IERs for all modes.

Table II
ANALYTICAL RELIABILITY ESTIMATION

Operation	Mode	$IER_{insn}^{op_faulty}$	p_{sobel}^{insn}	$p \cdot IER$
fetch		0.512	0.148	0.0760
decode		0.623	0.148	0.0924
alu_rrr_dc	**Total**	**0.199**	**0.148**	**0.0295**
	add	0.268	0.081	0.0218
	sub	0.133	0.010	0.0013
	and	0.064	1e-4	6e-6
	or	0.111	0.056	0.0062
	xor	0.169	0.002	0.0003
alu_rrr_ex	**Total**	**0.246**	**0.148**	**0.0547**
	add	0.256	0.081	0.0208
	sub	0.249	0.010	0.0024
	and	0.109	1e-4	1e-5
	or	0.232	0.056	0.0130
	xor	0.215	0.002	0.0003
writeback_dst		0.853	0.148	0.1265

2) AER: When $IER_{insn}^{op_faulty}$s for all operations and instructions are obtained from the testbenches we can use Equation 2 to estimate $AER_{op_faulty}^{app}$ based on the application profiling. Table III shows the estimation, experimental values and also relative deviation between both values averaged for three selected applications. In each experiment, one single bit fault with duration 1 clock cycle is injected randomly in time and location into the target operation. *All analytical reliability estimation values can be obtained through one single simulation which consumes negligible amount of time while each experimental value comes from 10000 LISA level fault simulation experiments*, which requires around $5 hours$ each for Sobel and FFT and $12 hours$ for IDCT. This is a significant improvement in the productivity and facilitates exploration by the application developer like the optimizations proposed in [16]. Naturally, for any change in the processor datapath or storage, the analytical model parameters need to be recomputed via benchmarking against instruction-set simulation-based or RTL-based reliability estimation flow.

Generally, for all three applications the estimated and experimental AER values of the same operation are close to each other. Regarding individual $AER_{op_faulty}^{app}$, *fetch*, *decode* and *writeback_dst* operations are apparently more vulnerable than the others since they reside on the paths of many operations. Besides, *address_generation* shows highest AER among all other operations, this happens since it is activated by *load* and *store* operations with direct access to the resources. *Nop* shows 0 error rates since it contributes nothing to the program execution. Compared among different applications, *ldc_ri_dc* is more vulnerable in FFT since coefficients are more frequently loaded in FFT than the others, while Sobel suffers more from faults in *alu_rri_dc* and *alu_rri_ex* since the compiler generates more assembly codes for calculation with immediate values.

For estimation accuracy, the results for operations with higher AER values show better matches. This happens since frequently

Table III
RELIABILITY ESTIMATION FOR SELECTED APPLICATIONS

Operation	$AER^{app}_{op_faulty}$						Rel. Dev.
	Sobel		FFT		IDCT		
	Est.	Exp.	Est.	Exp.	Est.	Exp.	
fetch	0.533	0.533	0.524	0.527	0.526	0.514	0.01
decode	0.629	0.635	0.595	0.606	0.610	0.616	0.01
writeback_dst	0.514	0.518	0.426	0.426	0.417	0.420	0.01
alu_rrr_dc	0.029	0.024	0.022	0.023	0.025	0.024	0.09
alu_rrr_ex	0.054	0.054	0.036	0.034	0.051	0.051	0.03
alu_rri_dc	0.041	0.043	0.020	0.018	0.018	0.018	0.05
alu_rri_ex	0.040	0.039	0.019	0.019	0.018	0.018	0.02
alu_rrri_dc	0.015	0.016	0.005	0.004	0.012	0.010	0.12
ld_rr_dc	0.074	0.070	0.056	0.055	0.083	0.078	0.05
address_gen	0.082	0.082	0.069	0.068	0.112	0.106	0.02
ld_mem	0.024	0.024	0.018	0.020	0.027	0.028	0.06
ldc_ri_dc	0.002	0.002	0.044	0.053	0.002	0.003	0.16
lui_ri_dc	0.003	0.005	0.004	0.005	0.003	0.004	0.35
st_rr_dc	0.026	0.025	0.024	0.024	0.042	0.041	0.02
st_mem	0.021	0.018	0.019	0.019	0.034	0.037	0.08
cmp_rr_dc	0.005	0.004	0.009	0.012	0.003	0.005	0.31
cmp_rr_ex	0.014	0.016	0.025	0.029	0.009	0.011	0.15
bra	0.025	0.018	0.041	0.035	0.031	0.028	0.17
branch_exe	0.011	0.011	0.021	0.021	0.011	0.015	0.11
branch_wb	0.012	0.012	0.014	0.014	0.015	0.015	4e-3
brau	0.023	0.023	0.024	0.023	0.022	0.024	0.04
brai	0.006	0.007	0.006	0.008	0.007	0.007	0.13
nop	0	0	0	0	0	0	0

called operations are more robust to the randomness during fault injection. Besides, AERs of operations which involve conditional behaviors such as *cmp_rr* and *bra* are highly dependent on the application characteristics, which make it difficult to predict from IERs obtained using a standard testbench.

In summary, the proposed analytical technique facilitates fast reliability estimation for the target processor architecture with sufficient accuracy compared with instruction-set simulation-based estimation.

V. CONCLUSION

Reliability has become a serious design concern with decreasing technology size. For processor designers, techniques to accurately and efficiently estimate the reliability are extremely important. In this work, a instruction-set simulation-based and an analytical reliability estimation techniques have been introduced. High-level fault injection contributes to the efficiency of the simulation based technique. The estimation accuracy of both the techniques are demonstrated through several embedded applications on a RISC processor and by benchmarking against an RTL-based reliability estimation flow.

Future work will include studying performance-tradeoffs against reliability by applying the proposed techniques. Besides, the fault injection technique will be extended to be used in system-level simulation.

REFERENCES

[1] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[2] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[3] D. Ernst, N. S. Kim, S. Das, S. Pant, R. R. Rao, T. Pham, C. H. Ziesler, D. Blaauw, T. M. Austin, K. Flautner, and Trevor N. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO*, pages 7–18, 2003.

[4] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *CHES*, pages 2–12, 2002.

[5] A. DeHon, H. M. Quinn, and N. P. Carter. Vision for cross-layer optimization to address the dual challenges of energy and reliability. In *DATE*, pages 1017–1022, 2010.

[6] J. Srinivasan, S. V. Adve, P. Bose, J. Rivers, and C. K. Hau. RAMP: A model for reliability aware microprocessor design. *IBM Research Report*, 2003.

[7] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walstra, and C. Dai. Impact of CMOS Scaling and SOI on soft error rates of logic processes. *VLSI Technology Digest of Technical Papers*, 2001.

[8] D. Kammler, J. Guan, G. Ascheid, R. Leupers and H. Meyr. A fast and flexible Platform for Fault Injection and Evaluation in Verilog-based Simulations. In *Proc. 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI '09)*, Shanghai, China, Jul 2009.

[9] M. Sugihara, T. Ishihara, K. Hashimoto and M. Muroyama. A Simulation-based Soft Error Estimation Methodology for Computer Systems. In *7th International Symposium on Quality Electronic Design, 2006*, march 2006.

[10] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *In Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2008.

[11] Synopsys. *Processor Designer* *http://www.synopsys.com/Systems/BlockDesign/processorDev*.

[12] Target Compiler Technologies. *http://www.target.com*.

[13] S. S. Mukherjee, C. T. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, pages 29–42, 2003.

[14] A. Biswas, P. Racunas, J. S. Emer, and S. S. Mukherjee. Computing Accurate AVFs using ACE Analysis on Performance Models: A Rebuttal. *Computer Architecture Letters*, 7(1):21–24, 2007.

[15] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. Reliable software for unreliable hardware: embedded code generation aiming at reliability. In *CODES+ISSS*, pages 237–246, 2011.

[16] S. Rehman, M. Shafique, F. Kriebel and J. Henkel. RAISE: Reliability-Aware Instruction SchEduling for unreliable hardware. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 671 –676, 30 2012-feb. 2 2012.

[17] S. Krishnaswamy, G. F. Viamontes, I. L. Markov and J. P. Hayes. Probabilistic transfer matrices in symbolic reliability analysis of logic circuits. *ACM Trans. Design Autom. Electr. Syst.*, 13(1), 2008.

[18] J. Lee and A. Shrivastava. Static Analysis of Register File Vulnerability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(4):607–616, 2011.

[19] A. Chattopadhyay, H. Meyr and R. Leupers. *LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation*, chapter 5, pages 95–130. Morgan Kaufmann, jun 2008.

[20] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: The mefisto tool. In *FTCS*, pages 66–75, 1994.

[21] J. Carlos Baraza, J. Gracia, D. Gil, and P. J. Gil. A prototype of a vhdl-based fault injection tool: description and application. *Journal of Systems Architecture*, 47(10):847–867, 2002.

[22] S. Krishnaswamy, G. F. Viamontes, I. L. Markov, and J. P. Hayes. Accurate Reliability Evaluation and Enhancement via Probabilistic Transfer Matrices. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 282–287, Washington, DC, USA, 2005. IEEE Computer Society.

[23] R. Leupers and O. Temam. *Processor and System-on-Chip Simulation*. Springer, 2010.